Scalable Montgomery Modular Multiplication Architecture with Low-Latency and Low-Memory Bandwidth Requirement

Wen-Ching Lin, Jheng-Hao Ye, and Ming-Der Shieh, *Member*, *IEEE*

Abstract—Montgomery modular multiplication is widely used in public-key cryptosystems. This work shows how to relax the data dependency in conventional word-based algorithms to maximize the possibility of reusing the current words of variables. With the greatly relaxed data dependency, we then proposed a novel scheduling scheme to alleviate the number of memory access in the developed scalable architecture. Analytical results show that the memory bandwidth requirement of the proposed scalable architecture is almost $1/(w-1)$ times that of conventional scalable architectures, where w denotes word size. The proposed one also retains a latency of exactly one cycle between the operations of the same words in two consecutive iterations of the Montgomery modular multiplication algorithm when employing enough processing elements. Compared to the design in the related work, experimental results demonstrate that the proposed one achieves an almost 54 percent reduction in power consumption with no degradation in throughput. The reduced number of memory access not only leads to lower power consumption, but also facilitates the design of scalable architectures for any precision of operands.

Index Terms—Cryptosystems, low-power design, Montgomery modular multiplication, scalable architecture, VLSI

1 Introduction

The Montgomery modular multiplication algorithm [1] is widely applied to public-key algorithms like Rivest-Shamir-Adleman (RSA) [2] and elliptic curve cryptography (ECC) [3], [4] for carrying out modular multiplication. Hardware implementations of Montgomery modular multiplication fall into two categories: one includes designs for fixed-precision input operands [5], [6], [7], in which full-precision multiplicand and modulus are processed while the multiplier is handled bit-by-bit, and the other includes scalable architectures for variable-precision input operands [8], [9], [10], [11], [12], in which the multiplicand and modulus are first divided into multiple words and then processed word-by-word with the multiplier processed bit-by-bit.

The first scalable architecture for Montgomery multiplication was proposed by Tenca and Koc [8], [9]. The data path can perform Montgomery modular multiplication with any precision. Moreover, the potential problem of high fan-out control signals is greatly relaxed since the word size is usually much smaller than the operand size. To reduce the latency of modular multiplication, multiple process elements (PEs) are employed to perform the operations in different iterations of the Montgomery modular multiplication algorithm concurrently. However, the latency between two neighboring PEs is two cycles, meaning that the minimum latency of the resulting architecture is almost twice that of designs for fixed precision, i.e., $2k$, where k is the modulus size. Thus, various kinds of schemes [10], [11], [12] have been proposed to achieve one-cycle latency between neighboring PEs. For convenience, Tenca's work and its improved designs are referred to conventional scalable architectures.

In conventional scalable architectures, one PE must access all words of the variables, including multiplicand, modulus, and intermediate results, to perform all the operations required for the current bit of the multiplier before moving to deal with the remaining bits of the multiplier. That is, from the algorithmic point of view, one PE cannot perform the operations in the next iteration (outer loop) until all the operations in the current iteration have completed. Since a variable is processed word-by-word, starting from the least significant word, for each bit of the multiplier, the current word may be overwritten by a newly accessed word to save the storage space and one word of a variable is read and/or written each cycle. Consequently, conventional scalable architectures demand high-memory bandwidth between memory, which stores multiplicand, modulus, and intermediate results, and the kernel (including PEs) to perform Montgomery modular multiplication. To improve performance, local memory is adopted in conventional scalable architectures [9]. It implies that the precisions of operands are limited by the internal memory size. Furthermore, high-memory bandwidth requirement results in large power consumption by local or external memory.

To reduce the memory bandwidth requirement, it is the most efficient that a PE in the kernel to access the jth word of variables once and to complete all operations of the jth word in all iterations of the Montgomery modular multiplication algorithm. However, an analysis of conventional Montgomery multiplication algorithms shows that the inherent right-shift operation results in the data

• *The authors are with the Department of Electrical Engineering, National Cheng Kung University, No.1, Ta-Hsueh Road, Tainan 70101, Taiwan. E-mail: {ioweiya, johnny}@vlsilab.ee.ncku.edu.tw, shiehm@mail.ncku.edu.tw.*

Manuscript received 6 Feb. 2012; revised 16 July 2012; accepted 20 Aug. 2012; published online 5 Sept. 2012.
Recommended for acceptance by P. Montuschi.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2012-02-0087.
Digital Object Identifier no. 10.1109/TC.2012.218.

dependency between the operation of the jth word in the $(i+1)$th iteration and that of the $(j+1)$th word in the ith iteration. As a result, a PE cannot interleave the operations of the jth word in the ith iteration with those of the same word in subsequent iterations in conventional Montgomery modular multiplication algorithm.

This present study modifies the conventional Montgomery multiplication algorithm to obtain a word-based Montgomery multiplication algorithm with low-memory bandwidth requirement. The low-memory bandwidth is achieved by relaxing the data dependency between the operation of the jth word in the $(i+1)$th iteration and that of the $(j+1)$th word in the ith iteration in the modified algorithm. In the proposed scalable architecture, a PE can interleave the operations of the jth word from the $(i+1)$th to the $(i+w-2)$th iterations and perform these operations sequentially before dealing with those of the $(j+1)$th word in the ith iteration, where w denotes word size. The jth words of the multiplicand, modulus, and intermediate results are accessed only once for all operations of the jth word from the $(i+1)$th to the $(i+w-2)$th iterations. As a result, memory bandwidth in the proposed scalable architecture is almost $1/(w-1)$ times that in conventional scalable architectures. Note that the benefit of reducing the number of memory access using the proposed scheme can also be achieved when multiple PEs are employed in the scalable architecture.

This work focuses on relaxing the data dependency in conventional word-based algorithm to maximize the possibility of reusing the current word of a variable for reducing the memory bandwidth requirement. With the greatly relaxed data dependency, we then propose a new scheduling scheme to alleviate the number of memory access in the developed scalable architecture. Compared to the design in [11], experimental results show that the proposed architecture achieves an almost 54 percent power consumption reduction with the same throughput as that in [11]. That is, when enough PEs are used, the feature of obtaining a latency of exactly one cycle between the operations of the jth word in the ith and $(i+1)$th iteration is also retained in the proposed architecture. The proposed technique can be extended to high-radix designs like those in [13], [14], as shown in [15], [16].

The rest of this paper is organized as follows: Section 2 briefly reviews the related modular multiplication algorithms and describes the notations used in this work. Section 3 presents the proposed word-based Montgomery modular multiplication algorithm. Section 4 shows the corresponding scalable architecture and experimental results. Section 5 concludes this work.

2 Background

2.1 Montgomery Modular Multiplication

The Montgomery modular multiplication algorithm [1] uses simple addition and shift operations to replace the time-consuming trial division in conventional modular multiplication. Let the modulus N be a k-bit odd number and a constant R be defined as $2^k \bmod N$. The N-residue of an integer A with respect to R is defined as $A \times R \bmod N$. Montgomery's algorithm for computing $A \times B \times R^{-1} \bmod N$ is stated as Algorithm *MM*(A, B, N), where A and B are integers smaller than N. The notation $B[i] \in \{0,1\}$ denotes the ith bit of B; thus, $B = \sum_{i=0}^{k-1} B[i]2^i$. Throughout this paper, the notation $X[i]$ denotes the ith bit of X in binary representation and $X[i:j]$ represents a segment of X from the ith to the jth bits.

Algorithm MM(A, B, N)
// The Montgomery modular multiplication algorithm
// *Inputs*: N (modulus, k bits), A (multiplicand, k bits), B (multiplier, k bits), where $A, B < N$
// *Output*: $S = A \times B \times R^{-1} \bmod N$, where $R \equiv 2^k \bmod N$ and $0 \le S < N$

```
{
  S = 0;
  for i = 0 to k − 1 {
    q = (S + A × B[i]) mod 2;
    S = (S + A × B[i] + q × N)/2;          (1)
  }
  if (S ≥ N) S = S − N;
 return S;
}
```

Since the convergence range of S is 0 to $2N$, an extra subtraction ($S = S - N$) is needed if $S \ge N$ at the end of Algorithm *MM*. When Algorithm *MM* is applied to modular exponentiation, which is performed by repeated modular multiplication, the final subtraction can be removed [17]. The recurrent equations in Algorithm *MM* are rewritten as

$$q^{(i)} = (S^{(i)} + A \times B[i]) \bmod 2, \tag{2}$$

$$S^{(i+1)} = (S^{(i)} + A \times B[i] + q^{(i)} \times N)/2, \tag{3}$$

for $i = 0$ to $k-1$ with $S^{(0)} = 0$, where the superscript denotes the iteration index.

Algorithm *MM* cannot directly perform modular multiplication when the size of the modulus is larger than k because the multiplicand A and modulus N are processed with full precision. Tenca's word-based Montgomery modular multiplication algorithm [9] is briefly reviewed below.

2.2 Word-Based Montgomery Modular Multiplication

In Tenca's algorithm [9], the k-bit modulus N and multiplicand A are partitioned into $\lceil k/w \rceil w$-bit words, where $\lceil x \rceil$ denotes the ceiling function of x. Let the subscript j denote the jth word and $e = \lceil k/w \rceil$. An operand can be expressed in terms of its partitioned words as

$$A = \sum_{j=0}^{e} A_j 2^{jw} = \sum_{j=0}^{e} A[(jw+w-1):jw]2^{jw},$$

where

$$A_j = \sum_{l=0}^{w-1} A_j[l]2^l = A[(jw+w-1):jw] = \sum_{l=0}^{w-1} A[jw+l]2^l$$

with $A[i] = 0$ for $i \ge k$. The notation $A_j[l]$ denotes the lth bit in the jth word of A. These expressions are used

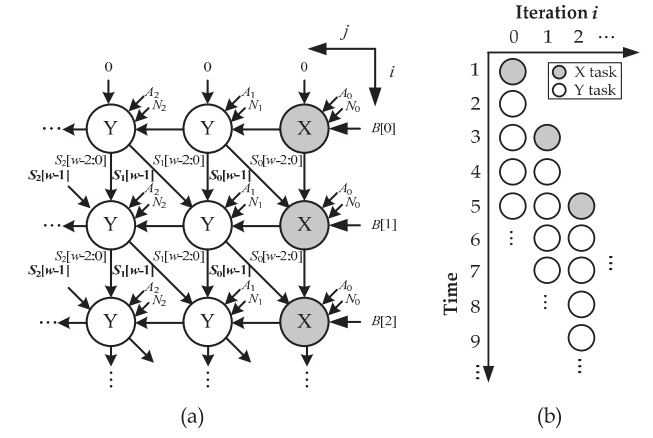

Fig. 1. (a) Dependency graph and (b) As soon as possible (ASAP) scheduling of Algorithm *Tenca_WBMM*.

interchangeably for partitioned words, such as A_j, N_j, and S_j, in this work. Note that when the intermediate result S is represented in binary form, the number of partitioned words becomes $e = \lceil (k+1)/w \rceil$, instead of $e = \lceil k/w \rceil$ in carry-save form. Without loss of generality, we use the notation e to denote either $\lceil k/w \rceil$ or $\lceil (k+1)/w \rceil$ where appropriate. Tenca's [9] word-based Montgomery modular multiplication algorithm is shown below.

Algorithm Tenca_WBMM(A, B, N)
// Word-based Montgomery modular multiplication
// *Inputs*: k-bit operands N, A, and B, where $A, B < N$
// *Output*: $S = A \times B \times R^{-1} \bmod N$,
where $R \equiv 2^k \bmod N$ and $0 \le S < 2N$

```
{
  S = 0;
  for i = 0 to k − 1{                                  //outer loop
    //X task: processing the least significant word
    (Ca, S_0) = S_0 + A_0 × B[i];
    q = S_0[0];
    (Cb, S_0) = S_0 + q × N_0;
    for j = 1 to e{                                    //inner loop
      //Y task: processing the other words
      (Ca, S_j) = S_j + A_j × B[i] + Ca;
      (Cb, S_j) = S_j + q × N_j + Cb;
      S_{j−1} = S_j[0] ∘ S_{j−1}[w − 1 : 1];                    (4)
    }
    S_e = 0;
  }
}
```

In the algorithm, the notation $X \circ Y$ denotes the concatenation of X and Y. Each iteration of the outer loop includes one X task for processing the least significant word and e tasks for processing the remaining words. As seen from (1) and (4), the right shifting of the intermediate result S in Algorithm *MM* is mapped to the word-based right shifting of $S_{j-1} = (S_j[0], S_{j-1}[w-1:1])$ in Algorithm *Tenca_WBMM*. Fig. 1a shows the dependency graph of Algorithm *Tenca_WBMM*, where i and j denote the iteration and word numbers, respectively. Equation (4) results in

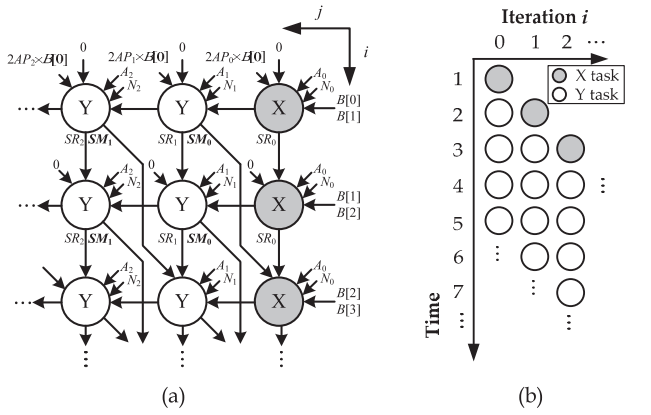

Fig. 2. (a) Dependency graph and (b) ASAP scheduling of the improved word-based Montgomery multiplication Algorithm [11].

that the delay time between two consecutive iterations is at least two cycles.

2.3 Modified Montgomery Modular Multiplication for Low-Latency Scalable Architecture

In [11], low latency is achieved by deferring the accumulation of the MSB of each word from the $(i+1)$th iteration in the original algorithm to the $(i+2)$th iteration. For operand reduction, the multiplicand A is decomposed into two components, AP and AR, such that $A = AP + AR$ with

$$AP[j] = \begin{cases} A[j] & \text{if } j = \tau w - 2 \\ 0 & otherwise \end{cases}$$
$$AR[j] = \begin{cases} A[j] & \text{if } j \neq \tau w - 2 \\ 0 & otherwise \end{cases} \tag{5}$$

for $\tau = 1$ to e. Let *SM* and *SR* denote the MSB and the rest part of each word of intermediate result S^* in the modified Algorithm *MM* in [11]; the corresponding recurrent equations can be written as

$$q^{(i)} = [(SR^{(i)} + 2AP \times B[i+1]) + (SM^{(i-1)}/2 + AR \times B[i])] \bmod 2 = \left(OP_1^{(i)} + OP_2^{(i)}\right) \bmod 2, \tag{6}$$

$$\begin{aligned}(SR^{(i+1)} + SM^{(i+1)}) &= [(SR^{(i)} + 2AP \times B[i+1]) \\ &\quad + (SM^{(i-1)}/2 + AR \times B[i]) \\ &\quad + q^{(i)} \times N]/2 \\ &= \left(OP_1^{(i)} + OP_2^{(i)} + OP_3^{(i)}\right)/2,\end{aligned} \tag{7}$$

where $OP_1^{(i)} = SR^{(i)} + 2AP \times B[i+1]$, $OP_2^{(i)} = SM^{(i-1)}/2 + AR \times B[i]$, and $OP_3^{(i)} = q^{(i)} \times N$ for $i = 0$ to $k-1$ with the initial values $SR^{(0)} = 0, SM^{(0)} = 0$, and $SM^{(-1)} = 2AP \times B[0]$. Note that a postprocessing operation, $S^{(k)} = SR^{(k)} + SM^{(k)} + SM^{(k-1)}/2$, is required to obtain the final result.

Equations (6) and (7) can also be implemented based on word-based operations as shown in Section 2.2. The corresponding dependency graph is shown in Fig. 2a. The delay time between the ith and $(i+1)$th iterations of the outer loop is exactly one cycle, as shown in Fig. 2b.

3 Proposed Word-Based Montgomery Modular Multiplication Algorithm

3.1 Data Dependency Relaxation for Low-Memory Bandwidth Requirement

In Tenca's and its improved scalable architectures, one PE will accomplish all the operations in the current iteration (outer loop) before starting to perform those in the remaining iterations. For simplicity of explanation, we assume that only one PE is employed in the design. The same conclusion also applies to the multiple PE solution. In each iteration, one PE must access again all words of the multiplicand, modulus, and intermediate results to perform the required operations. Since a variable is processed word-by-word, starting from the least significant word, for each bit of the multiplier, the current word is overwritten by a newly accessed word to save the storage space and one word of a variable is read and/or written each cycle. This implies that conventional scalable architectures demand high-memory bandwidth between memory, which stores multiplicand, modulus, and intermediate results, and the kernel (including PEs) to perform Montgomery modular multiplication.

To alleviate this problem, a feasible solution is that one PE can access the jth words of the variables and complete all the operations of the jth words in multiple iterations seamlessly, i.e., reusing the current words under processing to minimize the number of memory access. However, as shown in Fig. 1a, the MSB of S_j in the $(i+1)$th iteration is available when S_{j+1} in the ith iteration is processed. Thus, a PE cannot interleave the operations of the jth word in the ith iteration with those of the same word in subsequent iterations. As discussed in Section 2.3, our previous work [11] showed that the accumulation of the MSB of the jth word can be deferred to the next iteration because MSB of jth word is always divisible by 2 for $w \geq 2$. Actually, the MSB of jth word is always divisible by 2^{w-1} and, consequently, can be deferred to the $(w-1)$th iteration. We can take advantage of this property to reformulate Algorithm *MM* and then map the results to derive the desired word-based algorithm; a task in the word-based algorithm can perform the operations of the jth word in multiple iterations.

The algorithm modification and derivation is similar to that in [11] and, thus, not elaborated in this work. In contrast to (5), for operand reduction, the multiplicand A is first decomposed into two components, AP' with the LSB of each word and AR' with the remaining bits, such that $A = AP' + AR'$:

$$AP'[j] = \begin{cases} A[j] & \text{if } j = \tau w \\ 0 & \text{otherwise} \end{cases}$$
$$AR'[j] = \begin{cases} A[j] & \text{if } j \neq \tau w \\ 0 & \text{otherwise,} \end{cases} \tag{8}$$

for $\tau = 0$ to $e-1$. The reformulated recurrent equations can be written as

$$\begin{aligned} q^{(i)} &= [(SR'^{(i)} + 2^{w-1}AP' \times B[i+w-1]) \\ &\quad + (SM'^{(i-w+1)}/2^{w-1} + AR' \times B[i])] \bmod 2 \\ &= \left(OP'^{(i)}_1 + OP'^{(i)}_2\right) \bmod 2, \end{aligned} \tag{9}$$

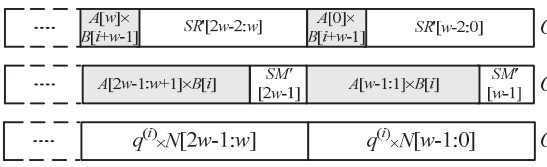

Fig. 3. Number of equivalent operands.

$$\begin{aligned} (SR'^{(i+1)} + SM'^{(i+1)}) &= [(SR'^{(i)} + 2^{w-1}AP' \times B[i+w-1]) \\ &\quad + (SM'^{(i-w+1)}/2^{w-1} + AR' \times B[i]) \\ &\quad + q^{(i)} \times N]/2 \\ &= \left(OP'^{(i)}_1 + OP'^{(i)}_2 + OP'^{(i)}_3\right)/2, \end{aligned} \tag{10}$$

where $OP'^{(i)}_1 = SR'^{(i)} + 2^{w-1} \times AP' \times B[i+w-1]$, $OP'^{(i)}_2 = SM'^{(i-w+1)}/2^{w-1} + AR' \times B[i]$, and $OP'^{(i)}_3 = q^{(i)} \times N$ for $i = 0$ to $k-1$ with the initial values $SR'^{(0)} = AP' \times \sum_{i=0}^{w-2} B[i]2^i$, $SM'^{(0)} = 0$ for $i \leq 0$. Note that from the hardware implementation point of view, the number of operands is three, as shown in Fig. 3. A postprocessing operation, $S^{(k)} = SR'^{(k)} + \sum_{i=0}^{w-1} SM'^{(k-i)}/2^i$, is required to obtain the final result. The modified Montgomery modular multiplication algorithm based on (9) and (10) is shown below.

Algorithm Modified_MM_R(A, B, N)
// Modified Montgomery modular multiplication algorithm with maximum data dependency relaxation
// *Inputs*: N (modulus, k bits), A (multiplicand, k bits), B (multiplier, k bits), where $A, B < N$; Note that (i) $AP'[j] = A[j]$ for $j = \tau w, 0 \leq \tau \leq e-1$, and $AP'[j] = 0$ otherwise; (ii) $AR'[j] = 0$ for $j = \tau w, 0 \leq \tau \leq e-1$, and $AR'[j] = A[j]$ otherwise.
// *Output*: $S = A \times B \times R^{-1} \bmod N$, where $R \equiv 2^k \bmod N$ and $0 \leq S < N$
{
$SM' = 0, SR' = AP' \times \sum_{i=0}^{w-2} B[i]2^i$,
$T(1) = T(2) = \cdots = T(w-1) = 0$;
for $i = 0$ to $k-1$ {
$q = ((SR' + 2^{w-1} \times AP' \times B[i+w-1])$
$+(SM' + AR' \times B[i])) \bmod 2$;
$S' = ((SR' + 2^{w-1} \times AP' \times B[i+w-1])$
$+(SM' + AR' \times B[i]) + q \times N)/2$;
$SM' = T(w-1)/2$;
for($j = w-1$ to 2){
$T(j) = T(j-1)/2$;
}
$T(1) = \sum_{j=0}^{e-1} S'[jw+w-1]2^{jw+w-1}$;
$SR\prime = \sum_{j=0}^{e-1} \sum_{l=0}^{w-2} S'[jw+l]2^{jw+l}$;
}
return $S = (T(1) + T(2) + \cdots + T(w-1) + SM') + SR'$;
}

Note that the variables $T(i)$ for $i = 1$ to $w-1$ are introduced to defer the accumulation of the MSB in each word to the next $(w-1)$th iteration, as indicated in (9) and (10).

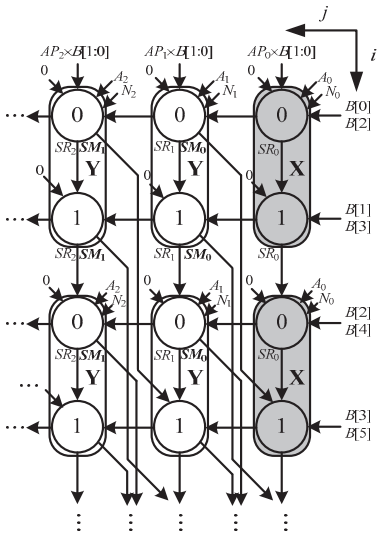

Fig. 4. Dependency graph of Algorithm *Modified_MM_R* for $w = 3$.

3.2 Proposed Word-Based Montgomery Modular Multiplication Algorithm

Fig. 4 illustrates the dependency graph of Algorithm *Modified_MM_R* for $w = 3$, where the jth word-based operation of the ith iteration is allocated in jth column and the ith row. For clarity, the symbols SM', SR', AP' are replaced by *SM*, *SR*, and *AP*, respectively. By taking advantage of that *SM* generated in the ith iteration is accumulated in the $(i + w)$th iteration, a task can perform all of the operations in $w - 1$ iterations instead of only one iteration. For example, since $w = 3$ in Fig. 4, one PE can execute all of the operations in two iterations in an interleaved manner. The two circles indexed with 0 and 1 in the same group, say X, denote that the two X tasks for the same words in two consecutive iterations can be accomplished before moving to handle the Y tasks for the next words of variables. A new interleaved word-based Montgomery modular multiplication algorithm called Algorithm *IWBMM_R* is proposed below.

Algorithm IWBMM_R(A, B, N)
// Proposed interleaved word-based Montgomery modular multiplication algorithm with maximum data dependency relaxation
// *Inputs*: N (modulus, k bits), A (multiplicand, k bits), B (multiplier, k bits), where $A, B < N$; Note that (i) $AP'[j] = A[j]$ for $j = \tau w, 0 \le \tau \le e - 1$, and $AP'[j] = 0$ otherwise; (ii) $AR'[j] = 0$ for $j = \tau w, 0 \le \tau \le e - 1$, and $AR'[j] = A[j]$ otherwise.
// *Output* : $S = A \times B \times R^{-1} \bmod N$, where $R \equiv 2^{\lceil k/(w-1) \rceil \times (w-1)} \bmod N$ and $0 \le S < 2N$
{
$SR' = AP' \times \sum_{i=0}^{w-2} B[i]2^i; SM' = 0;$
for $i_1 = 0$ to $\lceil k/(w-1) \rceil - 1$ {//outer loop
//X task: processing the least significant word
for $i_2 = 0$ to $w - 2$ {
$i = i_1 \times (w-1) + i_2;$
$(C_a[i_2], S'_0) = (SR'_0 + 2^{w-1} \times AP'_0 \times B[i + w - 1]) + (SM'_0[i_2] + AR'_0 \times B[i]);$
$q[i_2] = S'_0[0];$
$(C_b[i_2], S'_0) = S'_0 + q[i_2] \times N_0;$
$SR'_0 = S'_0/2;$
}
$SM'_0[0] = SM'_0[w-1];$
for $j = 1$ to e { //inner loop
//Y task: processing the other words
for $i_2 = 0$ to $w - 2$ {
$i = i_1 \times (w-1) + i_2;$
$(C_a[i_2], S'_j) = (SR'_j + 2^{w-1} \times AP'_j \times B[i + w - 1]) + (SM'_j[i_2] + AR'_j \times B[i]) + C_a[i_2];$
$(C_b[i_2], S'_j) = S'_j + q[i_2] \times N_j + C_b[i_2];$
$SR'_j = S'_j/2;$
$SM'_{j-1}[i_2 + 1] = S'_j[0];$
}
$SM'_j[0] = SM'_j[w-1];$
}
$SM'_e = 0;$
}
$C_c = 0;$
for $j = 0$ to e {//F task: Post-processing
$(C_c, S_j) = SR'_j + SM'_j + C_c;$
}
return S;
}

Each task X or Y has a loop with $(w - 1)$ iterations to sequentially perform all operations of the jth word in $(w - 1)$ iterations of Algorithm *IWBMM_R*. Thus, the outer loop has only $\lceil k/(w-1) \rceil$ iterations. $C_a[i_2]$ and $C_b[i_2]$ are the carry-out values in the i_2th iteration of X or Y task. Because of the deferred accumulation and inherent right-shift operation, the i_2th iteration in jth task generates 1 bit $SM_{j-1}[i_2 + 1]$, which is accumulated in the $(i_2 + 1)$th iteration of the $(j - 1)$th task in the next iteration of the outer loop for $i_2 \le (w - 2)$. SM_{j-1} [w] is accumulated in the first iteration of the $(j - 1)$th task in the $(i_2 + 2)$th iteration of the outer loop.

3.3 Scheduling and Performance Estimation

As stated in conventional scalable architectures, tasks in different iterations of the outer loop may be performed concurrently to improve performance. Fig. 5 shows the ASAP (as soon as possible) scheduling for Algorithm *IWBMM_R* with $w = 3$, in which the operations in the i_1th iteration of the outer loop are given in column i_1 and the tasks performed in time units from $2j - 1$ to $2j$ are arranged in rows $2j - 1$ to $2j$. There are $(e + 1)$ tasks in each column. The time unit is the latency incurred when accomplishing all operations in one iteration of a task. Each circle in Fig. 5 denotes the operations in one iteration of a task. The number inside a circle is i_2. The gray circles refer to the operations of X tasks. Each task has $(w - 1)$ circles. As shown in Fig. 5, A_j, N_j, SR_j, and SM_j are accessed only once for a PE to complete all operations in the jth task.

The number of PEs used for implementing Algorithm *IWBMM_R* can be adjusted for area/time tradeoffs. Assume that p PEs are employed; Figs. 6a and 6b show the schedules for $p \ge (e + 1)$ and $p < (e + 1)$, respectively. Note that each circle in Fig. 6 refers to a task for

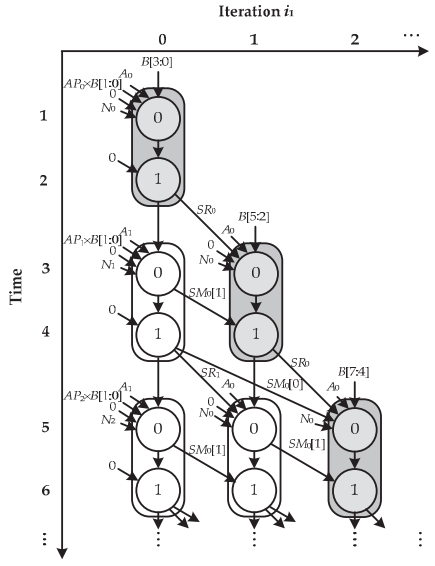

Fig. 5. ASAP scheduling of Algorithm *IWBMM_R* with $w = 3$.

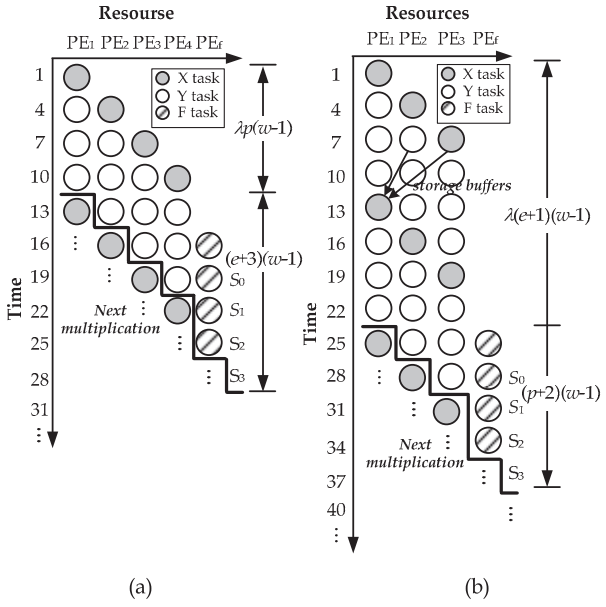

Fig. 6. Schedules for $k = 12$ and $w = 4$ with (a) $p = 4$ and (b) $p = 3$.

simplification. In the figures, the tasks in a column are assigned to the PE labeled at the top. For example, because PE_1, PE_2, and PE_3 in Fig. 6b are identical, they can be reused in multiple iterations of the outer loop. The symbol λ is defined as $\lceil k/(p(w-1))\rceil$; a PE needs to complete λ iterations of the outer loop to accomplish one Montgomery modular multiplication. As mentioned in [9], we can redefine $R = 2^{\lambda p(w-1)} \bmod N$ to simplify the hardware implementation because each task in Algorithm *IWBMM_R* performs $(w-1)$ iterations. $\mathrm{PE_f}$ is employed for postprocessing (F task), which produces the desired S by summation of SR' and SM'. Since PE_p generates SM'_{j-1} when the jth word is processed, the delay between PE_p and $\mathrm{PE_f}$ is two time units.

When the delay of all PEs is smaller than the time it takes to complete all $(e+1)$ tasks in an iteration of the outer loop, the intermediate results generated from the last two PEs (PE_p and PE_{p-1}) cannot be immediately processed by the first PE (PE_1). As shown in in Fig. 6b, PE_1 is still busy when the intermediate results are generated from PE_p and PE_{p-1}. The intermediate results must be, thus, stored in memory, denoted as storage buffers in Fig. 6b, until PE_1 is available. Therefore, the latency when p PEs are employed to perform k-bit Montgomery modular multiplication using the proposed algorithm can be expressed as (11). Moreover, as shown in Fig. 6, the consecutive Montgomery modular multiplications can be overlapped to increase performance; thus, the throughput can be estimated as (12):

$$L_O = \begin{cases} (\lambda p + e + 3))(w-1) & \text{if } (e+1) \le p \\ (\lambda(e+1) + p + 2)(w-1) & \text{otherwise,} \end{cases} \tag{11}$$

$$TP_O = \begin{cases} k/\lambda p(w-1) \approx 1 & \text{if } (e+1) \le p \\ k/(\lambda(e+1)(w-1)) \approx p/(e+1) & \text{otherwise.} \end{cases} \tag{12}$$

4 Hardware Architecture and Experimental Results

4.1 Hardware Design

To reduce the critical path delay, the intermediate results are kept in carry-save form and converted to binary form when Algorithm *IWBMM_R* ends. Fig. 7 shows the proposed scalable architecture derived based on Algorithm *IWBMM_R*. The architecture consists of two main blocks: kernel and shifter for multiplier. The multiplicand, modulus, multiplier, and/or intermediate results are stored in memory. Recall that the intermediate results are stored in storage buffers, if the condition $(e+1) > p$ holds. The scalable architecture accesses these variables through a memory interface. Generally speaking, the kernel is the main part of the interleaved word-based scalable architecture for carrying out the operations of Algorithm *IWBMM_R*, and the shifter is used to shift the multiplier, B, to the right by $p(w-1)$ bits.

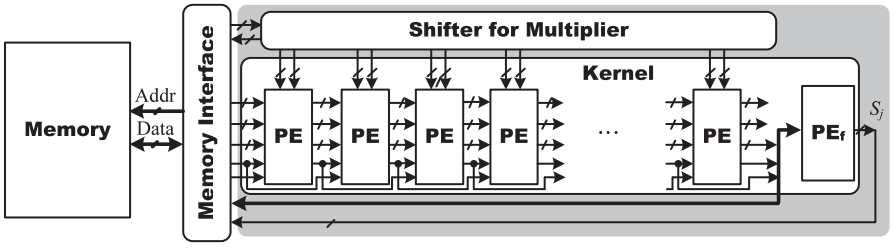

Fig. 7. Proposed interleaved scalable architecture of Montgomery modular multiplication.

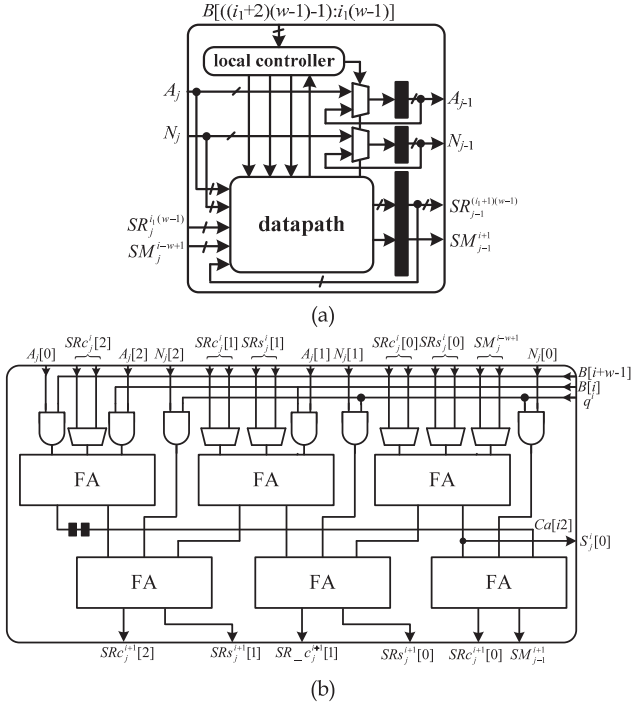

Fig. 8. (a) Simplified block diagram of PE; (b) illustration of data path design in each PE with $w = 3$.

The kernel consists of p PEs and one $\mathrm{PE_f}$. The PEs are designed for performing tasks X and Y; $\mathrm{PE_f}$ is employed for performing task F in Algorithm *IWBMM_R* and converting the final result from carry-save form into binary form. Fig. 8 shows a simplified block diagram of a PE and the data path design with $w = 3$, in which the black rectangle denotes flip-flops (FFs) and FA stands for a full adder. The symbol *Zc* (*Zs*) in Fig. 8 is used to denote the carry (sum) part of variable Z represented in carry-save form. Note that we use the symbol $SR_*(SM_*)$ instead of $SR'_*(SM'_*)$ for the intermediate results in the figure for clarity. It takes $(w-1)$ cycles to complete a task. As shown in Figs. 8a and 8b, each PE accesses variables SRc_j and SRs_j from the preceding PE or memory interface in the first cycle of each task. SRc_j and SRs_j are iteratively updated and stored in local FFs of the PE in each cycle of one task. In addition, each PE stores variables A_j and N_j in the local FFs for access by the following PE in the beginning of each task. When X task is processed, a local controller latches the control signals $B[((i_1+2)(w-1)-1):i_1(w-1)]$ and $(w-1)$-bit quotients for the following e Y tasks.

As mentioned above, the intermediate results are represented in carry-save form to reduce the critical path delay in the data path. The equations based on the carry-save representation can be directly extended from the operations, defined in binary form, in Algorithm *IWBMM_R*. A two-level adder tree is needed for the word-based operations, as also shown in [8], [9], [10], [11]. The main difference is that the $2w-1$ multiplexers are used for selecting $SRc'S$ and $SRs'S$ coming from the preceding PE in the first cycle of a task and local FFs in the following cycle of a task. One more multiplexer is employed for selecting *SM*'s generated from the preceding two PEs. $\mathrm{PE_f}$ includes w FAs for performing F tasks and a carry propagation adder (CPA) for format conversion. Since each PE outputs a word every $(w-1)$ cycles, the latency for format conversion is $(w-2)$ cycles assuming that one cycle is required for completing F task. Thus, a fast CPA is not needed in the proposed architecture.

TABLE 1
Average Memory Access per Cycle of Kernel Designs

	Read (bits)	Write (bits)
[9]	$2w(\sigma'+1)$	$2w\sigma'$
[11]	$2w(\sigma'+1)+\sigma'$	$(2w+1)\,\sigma'$
Proposed	$2w(\sigma+1)/(w-1)+\sigma$	$(2w/(w-1)+1)\,\sigma$

4.2 Experimental Results and Comparisons

Table 1 lists comparisons of memory bandwidth required for kernel designs in the proposed work and conventional scalable architectures [9], [11]. The symbols σ and σ' denote the average access of the variables stored in storage buffers of the proposed design and those in [9], [11], respectively, where $\lambda' = \lceil k/p \rceil$

$$\sigma = \begin{cases} \dfrac{\lambda-1}{\lambda} \approx 1 - p(w-1)/k & \text{if } (e+1) > p \\ 0 & \text{otherwise,} \end{cases} \tag{13}$$

$$\sigma' = \begin{cases} \dfrac{\lambda'-1}{\lambda'} \approx 1 - p/k & \text{if } (e+1) > Lp \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

Let L denote the latency between two neighboring PEs. Thus, [9, $L = 2$] and [11, $L = 1$]. In conventional scalable architectures, the kernel reads one word of the multiplicand (modulus) each cycle. In addition, assuming that carry-save form is employed, two words (carry and sum) of the intermediate results are read, except for the first iteration of the outer loop. The kernel needs to write two words of the intermediate results to storage buffers, except for the final iteration of the outer loop. One more bit in the design in [11] is read from or written to storage buffers because of the deferred accumulation used for reducing latency and increasing throughput. The proposed kernel design reads one word of the multiplicand (modulus) every $(w-1)$ cycles. Three words of the intermediate results are read from (written to) storage buffers every $(w-1)$ cycles, except for the first (final) iteration of the outer loop. As a result, the memory bandwidth is almost $1/(w-1)$ times those of the designs in [9], [11].

Algorithm *IWBMM_R* was coded in the Verilog hardware description language and synthesized using the Synopsys Design-Compiler based on TSMC 90-nm technology. The memory is static random access memory (SRAM) provided by Artisan compiler and an ideal memory interface is used. Note that the area of SRAM and the memory interface is not included in Table 2 The synthesis results of our work and the design [11] with $w = 16$ and $p = 69$ are listed in Table 2 and the power consumption levels estimated using Synopsys PrimeTime PX are shown in Fig. 9 The throughputs of the proposed design and the design in [11] are estimated using (12) and the following equation:

TABLE 2
Synthesis Results of Scalable Designs

		[11]	Proposed
Area (K gates)		82	103
Frequency (MHz)		595	595
Throughput (Mb/s)	1024 (e=64)	588	588
	2048 (e=128)	314	314
	3072 (e=192)	210	210
	4096 (e=256)	158	158
	8192 (e=512)	79	79
Memory Access (Mb/s)	1024 (e=64)	19,040	1,267
	2048 (e=128)	56,739	3,094
	3072 (e=192)	57,131	3,724
	4096 (e=256)	57,524	4,028
	8192 (e=512)	57,917	4,510

$$TP_{\mathrm{s}} = \begin{cases} k/(\lambda' p) \approx 1 & \text{if } (e+1) \le p \\ k/(\lambda'(e+1)) \approx p/(e+1) & \text{otherwise.} \end{cases} \quad (15)$$

The kernels of the proposed design and [11] can both operate at 724 MHz because the critical path delays of the two designs are dominated by the two-level adder tree, as shown in Fig. 8b. Note that the operating frequencies listed in Table 2 are limited by the employed SRAM with built-in self-test mechanism. As a result, the proposed design achieves almost the same throughput as that of the design in [11]. The throughput almost halves when e is doubled. This implies that more PEs can be used to increase throughput of 2,048-bit or larger Montgomery modular multiplication. The area of our work is larger than that of the design in [11] because of added multiplexers in Fig. 8b and more FFs in the local controller in Fig. 8a. The memory access, estimated based on Table 1 and operating frequency, in the proposed design is almost 0.05∼0.07 times that in the architecture [11].

As seen from Fig. 9, the power consumption of 1,024-bit modular multiplication for the proposed design is much smaller than those of the other designs because no data are read from and written to storage buffers. The power consumption increases slowly when the operand size is larger than 2,048 bits because of the small increase of σ and σ'. As shown in Fig. 9, the proposed design achieves significant power reduction. For example, when 2,048-bit modular

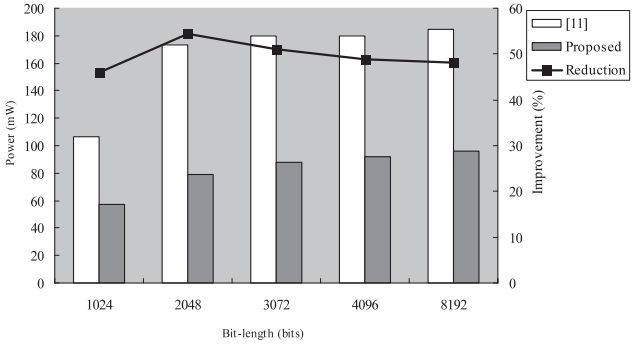

Fig. 9. Power consumption and reduction of Montgomery modular multiplications.

TABLE 3
Power Consumptions of the Designs for 2,048-Bit Montgomery Modular Multiplications

	[11]	Proposed	
	Power (mW)(1)	Power (mW) (2)	Reduction (%)†
SRAM	34.6	5.8	83.2
Random Logic	138.4	73.2	47.1
Total	173	79	54.3

†Reduction $= [(\mathrm{Power}^{(1)} - \mathrm{Power}^{(2)})/\mathrm{Power}^{(1)}] \times 100\%$.

Montgomery multiplication is performed, the reduction is 54.3 percent. Besides the low-memory bandwidth requirement between the kernel and memory module, reusing the current word of a variable in Algorithm *IWBMM_R* also results in a reduced number of transitions in pipelined registers for storing A_j and N_j in a PE. More specifically, the switching activity of pipelined registers in the proposed scalable architecture is $1/(w-1)$ times that of [11]. The values of power consumption reported from PrimeTime PX are listed in Table 3 for a 2,048-bit Montgomery modular multiplication. Experimental results show that the reduced number of memory access contributes about 16.6 percent $(0.85 \times 34.6/173)$ reduction in total power consumption, while the reduced switching activity of the kernel, in particular the pipelined registers, contributes another 37.7 percent power savings. Note that the reduced switching activity of the kernel is the outcome of employing the proposed scheduling scheme. The improvement resulting from low-memory bandwidth would be more significant when external memory is used in the design. Moreover, using external memory as the storage buffer can solve the limitation posed on increasing the operand precisions of the word-based modular Montgomery multiplication.

5 Conclusion

This work presented a technique for relaxing the data dependency in conventional word-based algorithms to maximize the possibility of reusing the current word of a variable. With the greatly relaxed data dependency and the proposed novel scheduling scheme, the number of memory access in the developed scalable architecture can be significantly reduced as compared to existing works. Moreover, the feature of obtaining a latency of exactly one cycle is also retained in the proposed architecture. Experimental results show that the proposed architecture achieves an almost 54 percent power consumption reduction with the same throughput in comparison with the design in the related work. The feature of low-memory bandwidth requirement facilitates the design of scalable architectures for any precision of operands. Finally, the proposed techniques can also be extended to various Montgomery modular multiplication algorithms, such as high-radix algorithms.

Acknowledgments

This work was supported in part by the National Science Council of R.O.C under Contract NSC 99-2221-E-006-221-MY3.

REFERENCES

[1] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation,* vol. 44, pp. 519-521, Apr. 1985.

[2] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM,* vol. 21, pp. 120-126, Feb. 1978.

[3] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Computation,* vol. 48, pp. 203-209, 1987.

[4] V.S. Miller, "Use of Elliptic Curve in Cryptography," *Proc. Advances in Cryptology (Crypto),* pp. 417-426, 1986.

[5] C.C. Yang, T.S. Chang, and C.W. Jen, "A New RSA Cryptosystem Hardware Design Based on Montgomery's Algorithm," *IEEE Trans. Circuits and Systems Part II: Analog and Digital Signal Processing,* vol. 45, no. 7, pp. 908-913, July 1998.

[6] C. McIvor, M. McLoone, and J.V. McCanny, "Modified Montgomery Modular Multiplication and RSA Exponentiation Techniques," *IEE Proc. Computer and Digital Techniques,* vol. 151, no. 6, pp. 402-408, Nov. 2004.

[7] M.D. Shieh, J.H. Chen, H.S. Wu, and W.C. Lin, "A New Modular Exponentiation Architecture for Efficient Design of RSA Cryptosystem," *IEEE Trans. Very Large Scale Integrated Systems,* vol. 16, no. 9, pp. 1151-1161, Sept. 2008.

[8] F. Tenca and K. Koc, "A Scalable Architecture for Montgomery Multiplication," *Proc. First Int'l Workshop Cryptographic Hardware and Embedded Systems (CHESS '99),* pp. 94-108, 1999.

[9] F. Tenca and K. Koc, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," *IEEE Trans. Computers,* vol. 52, no. 9, pp. 1215-1221, Sept. 2003.

[10] D. Harris, R. Krishnamurthy, S. Mathew, and S. Hsu, "An Improved Unified Scalable Radix-2 Montgomery Multiplier," *Proc. IEEE 17th Symp. Computer Arithmetic,* pp. 1196-1200, 2005.

[11] M.D. Shieh and W.C. Lin, "Word-Based Montgomery Modular Multiplication Algorithm for Low-Latency Scalable Architectures," *IEEE Trans. Computers,* vol. 59, no. 8, pp. 1145-1151, Aug. 2010.

[12] M. Huang, K. Gaj, and T. El-Ghazawi, "New Hardware Architectures for Montgomery Modular Multiplication Algorithm," *IEEE Trans. Computers,* vol. 60, no. 7, pp. 923-935, July 2011.

[13] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. IEEE 12th Symp. Computer Arithmetic,* pp. 193-199, 1995.

[14] P. Kornerip, "High-Radix Modular Multiplication for Cryptosystems," *Proc. IEEE 11th Symp. Computer Arithmetic,* pp. 277-283, 1993.

[15] F. Tenca, G. Todorov, and K. Koc, "High-Radix Design of a Scalable Modular Multiplier," *Proc. Third Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 189-205, 2001.

[16] N. Pinckney and D. Harris, "Parallelized Radix-4 Scalable Montgomery Multiplier," *Proc. 20th Ann. Conf. Integrated Circuits and Systems Design,* pp. 306-331, 2007.

[17] C.D. Walter, "Montgomery Exponentiation Needs no Final Subtractions," *Electronic Letters,* vol. 32, no. 21, pp. 1831-1832, Oct. 1999.

Wen-Ching Lin received the BS and MS degrees in electrical engineering from National Cheng Kung University in 2005 and 2007, respectively. He is currently working toward the PhD degree in electrical engineering from National Cheng Kung University, Taiwan, since 2007. His primary research areas are VLSI design and architecture, cryptography, finite field theory, and computer arithmetic.

Jheng-Hao Ye received the BS degree in electronic engineering from National Yunlin University of Science and Technology, Taiwan in 2008. He is currently working toward the PhD degree in electronic engineering from National Cheng Kung University, Taiwan. His primary research areas include VLSI design, cryptography architecture, and side channel analysis.

Ming-Der Shieh received the BS degree in electrical engineering from National Cheng Kung University, in 1984, the MS degree in electronic engineering from National Chiao Tung University, Taiwan, in 1986, and the PhD degree in electrical engineering from Michigan State University, East Lansing, in 1993. From 1988 to 1989, he was an engineer at United Microelectronic Corporation, Taiwan. From 1993 to 2002, he was with the faculty of the Department of Electronic Engineering, National Yunlin University of Science & Technology. He received the teaching award from NYUST in 1998 and was the Department chairman from 1999 to 2002. Since 2002, he has been with the Department of Electrical Engineering, National Cheng Kung University, where he is currently a full professor. His research interests include VLSI design and testing, VLSI for signal processing, and digital communication. He was the program cochair and the general cochair of Asian Test Symposium in 2004 and 2009, respectively, and the chairman of Tainan Chapter, IEEE Circuits and Systems Society from 2009 to 2010. He serves as the associated editor of *IEEE Transactions on Circuits and Systems - Part I* from 2010 to 2012, the lead guest editor of a special issue in *Journal Electrical and Computer Engineering* in 2012, and the technical committee member in several international conferences. In 2010, he joined Information and Communications Research Laboratories (ICL) of Industrial Technology Research Institute (ITRI) in Taiwan and is now the deputy general director of ICL. He is a member of the IEEE and the IEEE Computer Society.